

HOWTO zur Sortierung in Perl

(C) 2006 T.Birnthaler/H.Gottschalk <howtos(at)ostc.de>
OSTC GmbH, <http://www.ostc.de>

\$Id: perl-sorting-HOWTO.txt,v 1.10 2008-12-18 18:47:16 tsbirn Exp \$

Dieses Dokument beschreibt, wie in Perl das Sortieren von Arrays und Hashes durchgeführt wird.

Inhaltsverzeichnis

- 1) Einführung
 - 2) Absteigend und numerisch sortieren
 - 3) Hash sortieren
 - 4) Hierarchisch sortieren
 - 5) Schwartzsche Transformation
-

1) Einführung

Die Perl-Funktion "sort" sortiert Arrays standardmäßig "ASCII-alphabetisch aufsteigend" (etwa wie im Telefonbuch oder in einem Lexikon), indem sie die Ordnungsreihenfolge von jeweils 2 Elementen des Arrays mit Hilfe der String-Vergleichsoperation "cmp" (compare) ermittelt:

```
@sorted = sort @array;
```

Die Standard-Vergleichsfunktion "cmp" kann auch explizit als "Anonyme Funktion" direkt nach "sort" angegeben werden, folgender Aufruf ist also völlig identisch zu obigem:

```
@sorted = sort { $a cmp $b } @array;
```

Alternativ kann die Sortierfunktion explizit definiert und ihr Name beim "sort"-Aufruf mitgegeben werden. Folgender Aufruf ist also wiederum völlig identisch:

```
sub cmp_func { $a cmp $b }
@sorted = sort cmp_func @array;
```

Die zwei von Perl fest per Name "\$a" und "\$b" verwendeten Variablen zeigen dabei automatisch auf die 2 Elemente des Arrays, die gerade miteinander verglichen werden sollen (eigentlich sind es aus Performancegründen ALIASE für die Arrayelemente, also weitere Namen; eine Zuweisung an "\$a" oder "\$b" ändert also das entsprechende Element im Array --- keine gute Idee!).

Die Funktion "sort" ruft auf geschickte Art und Weise (Quicksort-Verfahren) solange die Vergleichs-Funktion mit je 2 Array-Elementen auf, bis ALLE Elemente gemäß der Vergleichsfunktion sortiert sind. Eine Element-Vergleichsfunktion muss folgende Werte zurückliefern:

```
1 falls "$a" größer "$b"
0 falls "$a" gleich "$b"
-1 falls "$a" kleiner "$b"
```

Will man zusehen, welche Elemente die Perl-Funktion "sort" in welcher Reihenfolge vergleicht, kann man eine "print"-Anweisung in die Sortierfunktion einbauen. Man sieht dann pro Aufruf der Sortierfunktion durch "sort" die beiden gerade miteinander verglichenen Elemente (man sieht auch, dass viele Elemente mehrfach an die Sortierfunktion übergeben werden).

```
sub cmp_func
{
    print "cmp_func: '$a' cmp '$b' -> ", $a cmp $b, "\n";
    $a cmp $b;
}
```

2) Absteigend und numerisch sortieren

Standardmäßig sortiert "sort" aufsteigend wie gesagt alphabetisch. Vertauscht man "\$a" und "\$b", so wird "umgekehrt" (d.h. normalerweise "ASCII-alphabetisch absteigend") sortiert, d.h. man spart den Aufruf der Funktion "reverse" ein.

```
@rev_sorted = sort { $b cmp $a } @array;          # Umgekehrt sortieren
@rev_sorted = reverse sort { $a cmp $b } @array;  # Analog (langsamer?)
```

Durch Austausch der Element-Vergleichsfunktion "cmp" gegen "<=>" kann auch "numerisch aufsteigend" (oder "absteigend") sortiert werden:

```
@sorted      = sort { $a <=> $b } @array;        # Numerisch aufsteigend
```

```
@rev_sorted = sort { $b <=> $a } @array; # Numerisch absteigend
```

3) Hash sortieren

Weiterhin ist auch die Sortierung von Hash-Schlüsseln nach ihrem zugehörigen Hash-Wert möglich, um auf die Hash-Elemente in einer nach den Werten sortierten Reihenfolge zugreifen zu können:

```
%hash = ("tom" => 17, "hans" => 35, "rick" => 5, "helmut" => 99);
# Variante A: Numerisch sortieren
@keys_sorted_by_value = sort { $hash{$a} <=> $hash{$b} } keys %hash;
# Variante B: Textuell sortieren
@keys_sorted_by_value = sort { $hash{$a} cmp $hash{$b} } keys %hash;
foreach (@keys_sorted_by_value) {
    print "$_ hat Wert $hash{$_}\n";
}
```

Der Trick ist hier, die Vergleichsfunktion für 2 Schlüssel auf Basis der ihnen zugeordneten Werte zu definieren. D.h. bei jedem Vergleich zweier Schlüssel wird auf die ihnen zugeordneten Werte zugegriffen. Also werden die Schlüssel gemäß den ihnen zugeordneten Werten sortiert.

4) Hierarchisch sortieren

Sogar nach MEHREREN Kriterien kann HIERARCHISCH sortiert werden (hier erst OHNE Beachtung der GROSS/Kleinschreibung, wenn die Elemente dabei gleich sind, dann MIT Beachtung der GROSS/Kleinschreibung):

```
@sorted = sort { uc($a) cmp uc($b) or $a cmp $b } @array; # uc=uppercase
@sorted = sort { "\L$a" cmp "\L$b" or $a cmp $b } @array; # lc=lowercase
```

Liefert ein Kriterium keinen Unterschied (Wert "0"), dann wird aufgrund der "or"-Verknüpfung nach dem nächsten Kriterium sortiert. Liefert ein Kriterium einen Unterschied, dann bricht die Auswertung aufgrund der "Shortcut/Short circuit"-Eigenschaft der Auswertung von Booleschen Ausdrücken an dieser Stelle mit dem Ergebnis dieses Vergleichs ("1" oder "-1") ab.

Man sollte in einer Vergleichs-Funktion im letzten Vergleichs-Schritt immer die Elemente direkt vergleichen, damit eine "eindeutige" Sortierung entsteht, die bei wiederholter Sortierung reproduziert wird:

```
sub cmp_2_levels {
    $alter{$a} <=> $alter{$b} or
    $a cmp $b;
}
sort cmp_2_levels keys %alter;
```

Im letzten Beispiel werden Benutzer-IDs einer Bibliothek gemäß einiger Sortierkriterien sortiert. Ist bzgl. eines Kriteriums kein Unterschied vorhanden, wird das nächste betrachtet (bis am Schluss die Benutzer-IDs direkt verglichen werden, die auf jeden Fall unterschiedlich sind):

- 1) Gezahlte Gebühr (Funktionsaufruf "&gebuehr" mit Benutzer-ID)
- 2) Anzahl geliehener Bücher (Hash "%buecher")
- 3) Nachname (Hash "%name")
- 4) Vorname (Hash "%vorname")
- 5) Benutzer-ID

```
@benutzer_ids = sort {
    &gebuehr($a) <=> &gebuehr($b) or
    $buecher{$a} <=> $buecher{$b} or
    $name{$a} cmp $name{$b} or
    $vorname{$a} cmp $vorname{$b} or
    $a <=> $b
} @benutzer_ids
```

5) Schwartzsche Transformation

Ein vom Perl-Guru Randal L. Schwartz erfundener Trick (bzw. inzwischen ein sogenanntes "Perl Idiom"), um eine Liste von "Dingen" gemäß einem Kriterium zu sortieren, das eine recht "teure" Funktion des "Dings" ist (d.h. das pro "Ding" aufwendig zu berechnen ist). Beispielsweise ist die Sortierung einer Liste von Dateien nach ihrem Alter (-M = modification time) sehr langsam, da pro Vergleich zwei Dateisystemzugriffe erfolgen:

```
@sortiert = sort { -M $a <=> -M $b } @dateien;
```

Grund: Gemäß dem weiter oben beschriebenen Verfahren, wie "sort" arbeitet, wird jedes Element eines Arrays mehrfach an die Sortierfunktion übergeben,

die Berechnung des Sortierkriteriums erfolgt also "mehrfach pro Ding".

Diesen Aufwand kann man auf den minimal notwendigen Aufwand reduzieren, indem man das Sortierkriterium pro "Ding" genau 1x berechnet, zusammen mit dem "Ding" zwischenspeichert und die Sortierung der "Dinge" auf Basis dieser Zwischenwerte durchführt. Man opfert also Speicherplatz, um Laufzeit zu sparen.

Dazu wandelt man das ursprüngliche 1-dimensionale Array "@dateien" in ein 2-dimensionales Array um, das aus einer Liste von (anonymen) Array-Referenzen auf die Paare ("Ding", Sortierkriterium) besteht. Pro "Ding" wird dabei 1x das Sortierkriterium berechnet (hier das Dateialter):

```
@refliste = map { [ $_, -M ] } @dateien;
```

Danach sortiert man diese Referenzen nach dem darin als 2-tes Element gespeicherten Sortierkriterium:

```
@refsortiert = sort { $a->[1] <=> $b->[1] } @refliste;
```

Und am Schluss wirft man das Sortierkriterium wieder weg und behält nur die ursprünglichen "Dinge" wieder übrig:

```
@sortiert = map { $_->[0] } @refsortiert;
```

Diese 3 Schritte führt man normalerweise ohne die obigen Zwischenvariablen @refliste und @refsortiert "auf einen Schlag" durch und nennt das Ganze dann nach ihrem Erfinder "Schwartzsche Transformation":

```
@sortiert = map { $_->[0] }  
             sort { $a->[1] <=> $b->[1] }  
             map { [ $_, -M ] } @dateien;
```

Genial, oder?